主讲老师： Fox

课前须知：

重要的事强调三遍：

1. 学习本节课，务必先学习上节课《**9. 深入理解AQS之独占锁ReentrantLock源码分析**》
2. 跟AQS实现的工具类源码前，务必先跟一下ReentrantLock的源码

并发知识体系脑图：

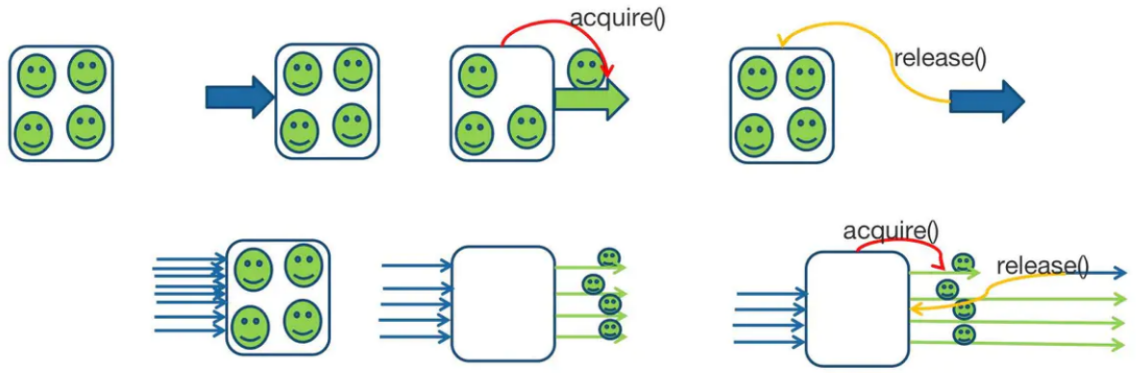
https://www.processon.com/view/link/615d4a610e3e74663e97fa0e#map

有道云链接：http://note.youdao.com/noteshare?id=acc718b75f6db2a11de6a6eceb4c9aa1&sub=C8E7777627C7472AA3086DDAC9D6945E

Semaphore介绍

Semaphore，俗称信号量，它是操作系统中PV操作的原语在java的实现，它也是基于AbstractQueuedSynchronizer实现的。

Semaphore的功能非常强大，大小为1的信号量就类似于互斥锁，通过同时只能有一个线程获取信号量实现。大小为n（n>0）的信号量可以实现限流的功能，它可以实现只能有n个线程同时获取信号量。



> PV操作是操作系统一种实现进程互斥与同步的有效方法。PV操作与信号量（S）的处理相关，P表示通过的意思，V表示释放的意思。用PV操作来管理共享资源时，首先要确保PV操作自身执行的正确性。
> P操作的主要动作是：
> ①S减1；
> ②若S减1后仍大于或等于0，则进程继续执行；
> ③若S减1后小于0，则该进程被阻塞后放入等待该信号量的等待队列中，然后转进程调度。
> V操作的主要动作是：
> ①S加1；
> ②若相加后结果大于0，则进程继续执行；
> ③若相加后结果小于或等于0，则从该信号的等待队列中释放一个等待进程，然后再返回原进程继续执行或转进程调度。

Semaphore 常用方法

构造器

```java
public Semaphore(int permits) {
    sync = new NonfairSync(permits);
}

/**...*/
public Semaphore(int permits, boolean fair) {
    sync = fair ? new FairSync(permits) : new NonfairSync(permits);
}
```

- permits 表示许可证的数量（资源数）
- fair 表示公平性，如果这个设为 true 的话，下次执行的线程会是等待最久的线程

常用方法

```java
public void acquire() throws InterruptedException
public boolean tryAcquire()
public void release()
public int availablePermits()
public final int getQueueLength()
public final boolean hasQueuedThreads()
protected void reducePermits(int reduction)
protected Collection<Thread> getQueuedThreads()
```

- acquire() 表示阻塞并获取许可
- tryAcquire() 方法在没有许可的情况下会立即返回 false，要获取许可的线程不会阻塞
- release() 表示释放许可
- int availablePermits()：返回此信号量中当前可用的许可证数。
- int getQueueLength()：返回正在等待获取许可证的线程数。
- boolean hasQueuedThreads()：是否有线程正在等待获取许可证。
- void reducePermit（int reduction）：减少 reduction 个许可证
- Collection getQueuedThreads()：返回所有等待获取许可证的线程集合

应用场景

可以用于做流量控制，特别是公用资源有限的应用场景

限流

```java
public class SemaphoneTest2 {

 /**
 * 实现一个同时只能处理5个请求的限流器
 */
 private static Semaphore semaphore = new Semaphore(5);

 /**
 * 定义一个线程池
```

```
10   */
11   private static ThreadPoolExecutor executor = new ThreadPoolExecutor(10, 50, 6
0, TimeUnit.SECONDS, new LinkedBlockingDeque<>(200));
12
13   /**
14   * 模拟执行方法
15   */
16   public static void exec() {
17   try {
18   semaphore.acquire(1);
19   // 模拟真实方法执行
20   System.out.println("执行exec方法" );
21   Thread.sleep(2000);
22   } catch (Exception e) {
23   e.printStackTrace();
24   } finally {
25   semaphore.release(1);
26   }
27   }
28
29   public static void main(String[] args) throws InterruptedException {
30   {
31   for (; ; ) {
32   Thread.sleep(100);
33   // 模拟请求以10个/s的速度
34   executor.execute(() -> exec());
35   }
36   }
37   }
38 }
```

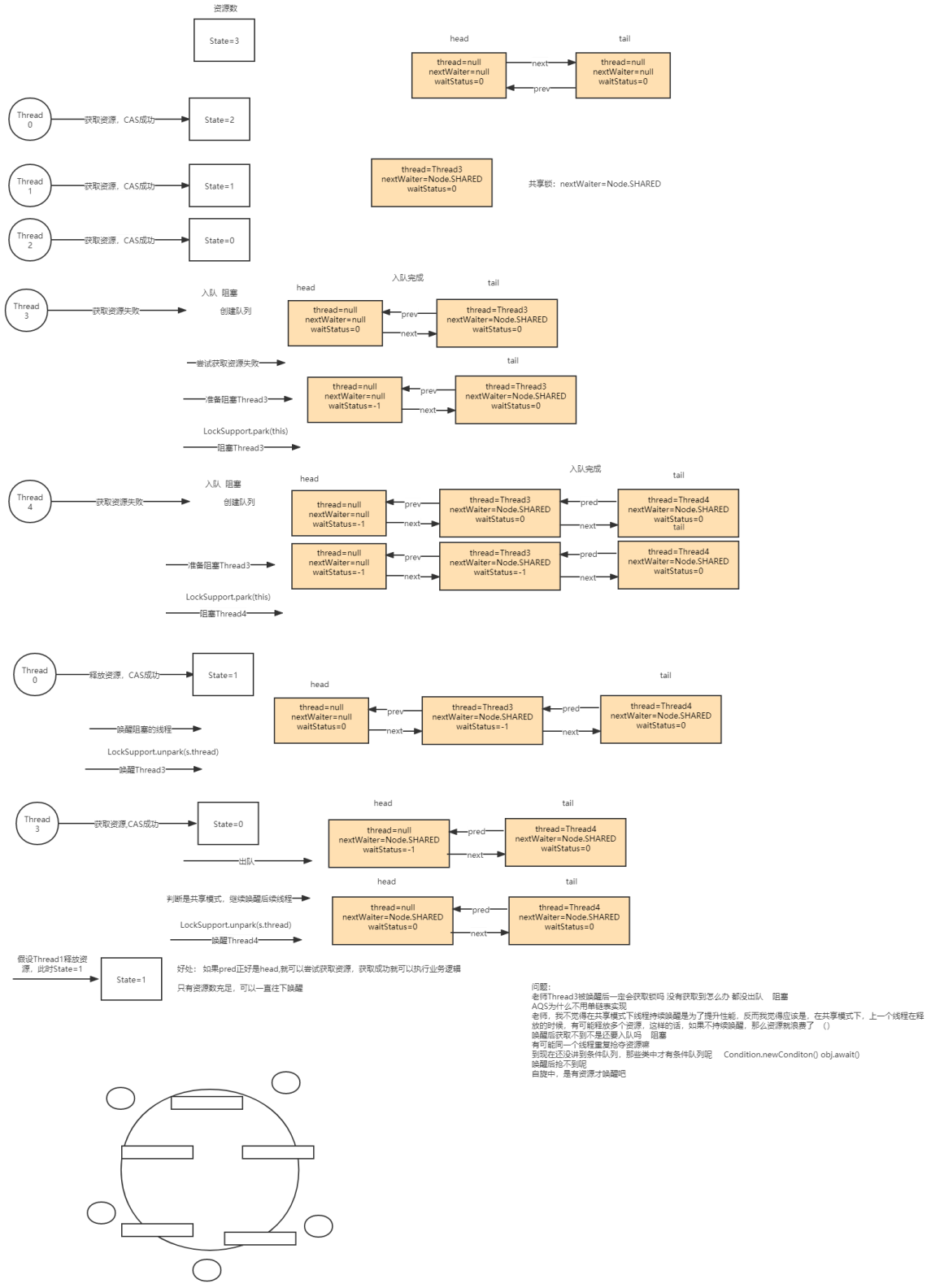
Semaphore源码分析

建议： 先跟上节课ReentrantLock的源码，再来跟Semaphore的源码

关注点：

1. Semaphore的加锁解锁（共享锁）逻辑实现
2. 线程竞争锁失败入队阻塞逻辑和获取锁的线程释放锁唤醒阻塞线程竞争锁的逻辑实现

https://www.processon.com/view/link/61950f6e5653bb30803c5bd2

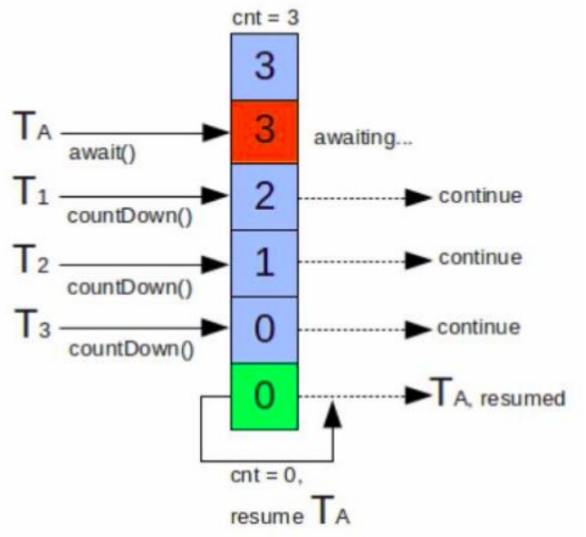


CountDownLatch介绍

CountDownLatch（闭锁）是一个同步协助类，允许一个或多个线程等待，直到其他线程完成操作集。

CountDownLatch使用给定的计数值（count）初始化。await方法会阻塞直到当前的计数值（count）由于countDown方法的调用达到0，count为0之后所有等待的线程都会被释放，并且

随后对await方法的调用都会立即返回。这是一个一次性现象 —— count不会被重置。如果你需要一个重置count的版本，那么请考虑使用CyclicBarrier。



CountDownLatch的使用

构造器

```
public CountDownLatch(int count) {
    if (count < 0) throw new IllegalArgumentException("count < 0");
    this.sync = new Sync(count);
}
```

常用方法

```
1  // 调用 await() 方法的线程会被挂起，它会等待直到 count 值为 0 才继续执行
2  public void await() throws InterruptedException { };
3  // 和 await() 类似，若等待 timeout 时长后，count 值还是没有变为 0，不再等待，继续执行
4  public boolean await(long timeout, TimeUnit unit) throws InterruptedException { };
5  // 会将 count 减 1，直至为 0
6  public void countDown() { };
```

CountDownLatch应用场景

CountDownLatch一般用作多线程倒计时计数器，强制它们等待其他一组（CountDownLatch的初始化决定）任务执行完成。

CountDownLatch的两种使用场景：

- 场景1：让多个线程等待
- 场景2：让单个线程等待。

场景1 让多个线程等待：模拟并发，让并发线程一起执行

```
1  public class CountDownLatchTest {
2
3   public static void main(String[] args) throws InterruptedException {
```

```
CountDownLatch countDownLatch = new CountDownLatch(1);
for (int i = 0; i < 5; i++) {
new Thread(() -> {
try {
//准备完毕……运动员都阻塞在这，等待号令
countDownLatch.await();
 String parter = "【" + Thread.currentThread().getName() + "】";
 System.out.println(parter + "开始执行……");
 } catch (InterruptedException e) {
 e.printStackTrace();
 }
 }).start();
 }

 Thread.sleep(2000);// 裁判准备发令
 countDownLatch.countDown();// 发令枪：执行发令
 }
}
```

场景2 让单个线程等待：多个线程(任务)完成后，进行汇总合并

很多时候，我们的并发任务，存在前后依赖关系；比如数据详情页需要同时调用多个接口获取数据，并发请求获取到数据后、需要进行结果合并；或者多个数据操作完成后，需要数据check；这其实都是：在多个线程(任务)完成后，进行汇总合并的场景。

```
public class CountDownLatchTest2 {
 public static void main(String[] args) throws Exception {

 CountDownLatch countDownLatch = new CountDownLatch(5);
 for (int i = 0; i < 5; i++) {
 final int index = i;
 new Thread(() -> {
 try {
 Thread.sleep(1000 + ThreadLocalRandom.current().nextInt(1000));
 System.out.println(Thread.currentThread().getName()+" finish task" + index );

 countDownLatch.countDown();
 } catch (InterruptedException e) {
 e.printStackTrace();
 }
 }).start();
 }

 // 主线程在阻塞，当计数器==0，就唤醒主线程往下执行。
```

```
20    countDownLatch.await();
21    System.out.println("主线程:在所有任务运行完成后，进行结果汇总");
22
23    }
24  }
```

CountDownLatch实现原理

底层基于 AbstractQueuedSynchronizer 实现，CountDownLatch 构造函数中指定的count直接赋给AQS的state；每次countDown()则都是release(1)减1，最后减到0时unpark阻塞线程；这一步是由最后一个执行countdown方法的线程执行的。

而调用await()方法时，当前线程就会判断state属性是否为0，如果为0，则继续往下执行，如果不为0，则使当前线程进入等待状态，直到某个线程将state属性置为0，其就会唤醒在await()方法中等待的线程。

CountDownLatch与Thread.join的区别

- CountDownLatch的作用就是允许一个或多个线程等待其他线程完成操作，看起来有点类似join() 方法，但其提供了比 join() 更加灵活的API。
- CountDownLatch可以手动控制在n个线程里调用n次countDown()方法使计数器进行减一操作，也可以在一个线程里调用n次执行减一操作。
- 而 join() 的实现原理是不停检查join线程是否存活，如果 join 线程存活则让当前线程永远等待。所以两者之间相对来说还是CountDownLatch使用起来较为灵活。

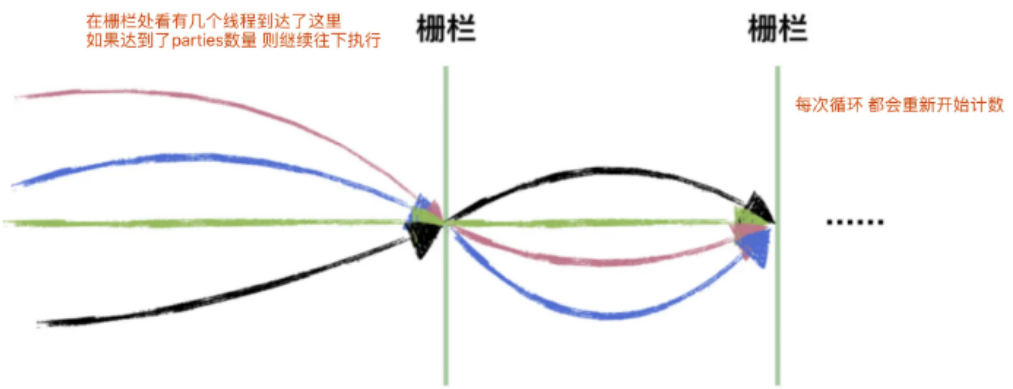
CountDownLatch与CyclicBarrier的区别

CountDownLatch和CyclicBarrier都能够实现线程之间的等待，只不过它们侧重点不同：

1. CountDownLatch的计数器只能使用一次，而CyclicBarrier的计数器可以使用reset() 方法重置。所以CyclicBarrier能处理更为复杂的业务场景，比如如果计算发生错误，可以重置计数器，并让线程们重新执行一次
2. CyclicBarrier还提供getNumberWaiting(可以获得CyclicBarrier阻塞的线程数量)、isBroken(用来知道阻塞的线程是否被中断)等方法。
3. CountDownLatch会阻塞主线程，CyclicBarrier不会阻塞主线程，只会阻塞子线程。
4. CountDownLatch和CyclicBarrier都能够实现线程之间的等待，只不过它们侧重点不同。CountDownLatch一般用于一个或多个线程，等待其他线程执行完任务后，再执行。CyclicBarrier一般用于一组线程互相等待至某个状态，然后这一组线程再同时执行。
5. CyclicBarrier 还可以提供一个 barrierAction，合并多线程计算结果。
6. CyclicBarrier是通过ReentrantLock的"独占锁"和Conditon来实现一组线程的阻塞唤醒的，而CountDownLatch则是通过AQS的“共享锁”实现

CyclicBarrier介绍

字面意思回环栅栏（循环屏障），通过它可以实现让一组线程等待至某个状态（屏障点）之后再全部同时执行。叫做回环是因为当所有等待线程都被释放以后，CyclicBarrier可以被重用。



CyclicBarrier的使用

构造方法

```
1  // parties表示屏障拦截的线程数量，每个线程调用 await 方法告诉 CyclicBarrier 我已经到达了屏障，然后当前线程被阻塞。
2  public CyclicBarrier(int parties)
3  // 用于在线程到达屏障时，优先执行 barrierAction，方便处理更复杂的业务场景(该线程的执行时机是在到达屏障之后再执行)
4  public CyclicBarrier(int parties, Runnable barrierAction)
```

重要方法

```
1 //屏障 指定数量的线程全部调用await()方法时，这些线程不再阻塞
2 // BrokenBarrierException 表示栅栏已经被破坏，破坏的原因可能是其中一个线程 await() 时被中断或者超时
3 public int await() throws InterruptedException, BrokenBarrierException
4 public int await(long timeout, TimeUnit unit) throws InterruptedException, BrokenBarrierException, TimeoutException
5
6 //循环 通过reset()方法可以进行重置
7 public void reset()
```

CyclicBarrier应用场景

CyclicBarrier 可以用于多线程计算数据，最后合并计算结果的场景。

```
1 public class CyclicBarrierTest2 {
2
3  //保存每个学生的平均成绩
4  private ConcurrentHashMap<String, Integer> map=new ConcurrentHashMap<String,Integer>();
5
6  private ExecutorService threadPool= Executors.newFixedThreadPool(3);
7
```

```
8  private CyclicBarrier cb=new CyclicBarrier(3,()->{
9  int result=0;
10  Set<String> set = map.keySet();
11  for(String s:set){
12  result+=map.get(s);
13  }
14  System.out.println("三人平均成绩为:"+(result/3)+"分");
15  });
16
17
18  public void count(){
19  for(int i=0;i<3;i++){
20  threadPool.execute(new Runnable(){
21
22  @Override
23  public void run() {
24  //获取学生平均成绩
25  int score=(int)(Math.random()*40+60);
26  map.put(Thread.currentThread().getName(), score);
27  System.out.println(Thread.currentThread().getName()
28  +"同学的平均成绩为: "+score);
29  try {
30  //执行完运行await(),等待所有学生平均成绩都计算完毕
31  cb.await();
32  } catch (InterruptedException | BrokenBarrierException e) {
33  e.printStackTrace();
34  }
35  }
36
37  });
38  }
39  }
40
41  public static void main(String[] args) {
42  CyclicBarrierTest2 cb=new CyclicBarrierTest2();
43  cb.count();
44  }
45 }
```

利用CyclicBarrier的计数器能够重置，屏障可以重复使用的特性，可以支持类似“人满发车”的场景

```
1 public class CyclicBarrierTest3 {
2
3  public static void main(String[] args) {
```

```
4
5   AtomicInteger counter = new AtomicInteger();
6   ThreadPoolExecutor threadPoolExecutor = new ThreadPoolExecutor(
7   5, 5, 1000, TimeUnit.SECONDS,
8   new ArrayBlockingQueue<>(100),
9   (r) -> new Thread(r, counter.addAndGet(1) + " 号 "),
10   new ThreadPoolExecutor.AbortPolicy());
11
12   CyclicBarrier cyclicBarrier = new CyclicBarrier(5,
13   () -> System.out.println("裁判：比赛开始~~"));
14
15   for (int i = 0; i < 10; i++) {
16   threadPoolExecutor.submit(new Runner(cyclicBarrier));
17   }
18
19   }
20   static class Runner extends Thread{
21   private CyclicBarrier cyclicBarrier;
22   public Runner (CyclicBarrier cyclicBarrier) {
23   this.cyclicBarrier = cyclicBarrier;
24   }
25
26   @Override
27   public void run() {
28   try {
29   int sleepMills = ThreadLocalRandom.current().nextInt(1000);
30   Thread.sleep(sleepMills);
31   System.out.println(Thread.currentThread().getName() + " 选手已就位，准备共用
时：  " + sleepMills + "ms" + cyclicBarrier.getNumberWaiting());
32   cyclicBarrier.await();
33
34   } catch (InterruptedException e) {
35   e.printStackTrace();
36   }catch(BrokenBarrierException e){
37   e.printStackTrace();
38   }
39   }
40   }
41
42  }
```

```
4 号  选手已就位，准备共用时：  197ms0
5 号  选手已就位，准备共用时：  527ms1
1 号  选手已就位，准备共用时：  740ms2
2 号  选手已就位，准备共用时：  775ms3
3 号  选手已就位，准备共用时：  796ms4
裁判：比赛开始~~
4 号  选手已就位，准备共用时：  702ms0
5 号  选手已就位，准备共用时：  897ms1
3 号  选手已就位，准备共用时：  925ms2
2 号  选手已就位，准备共用时：  940ms3
1 号  选手已就位，准备共用时：  956ms4
裁判：比赛开始~~
```

CyclicBarrier与CountDownLatch的区别

7. CountDownLatch的计数器只能使用一次，而CyclicBarrier的计数器可以使用reset() 方法重置。所以CyclicBarrier能处理更为复杂的业务场景，比如如果计算发生错误，可以重置计数器，并让线程们重新执行一次

8. CyclicBarrier还提供getNumberWaiting(可以获得CyclicBarrier阻塞的线程数量)、isBroken(用来知道阻塞的线程是否被中断)等方法。

9. CountDownLatch会阻塞主线程，CyclicBarrier不会阻塞主线程，只会阻塞子线程。

10. CountDownLatch和CyclicBarrier都能够实现线程之间的等待，只不过它们侧重点不同。CountDownLatch一般用于一个或多个线程，等待其他线程执行完任务后，再执行。CyclicBarrier一般用于一组线程互相等待至某个状态，然后这一组线程再同时执行。

11. CyclicBarrier 还可以提供一个 barrierAction，合并多线程计算结果。

12. CyclicBarrier是通过ReentrantLock的"独占锁"和Conditon来实现一组线程的阻塞唤醒的，而CountDownLatch则是通过AQS的“共享锁”实现

CyclicBarrier源码分析

关注点：

1.一组线程在触发屏障之前互相等待，最后一个线程到达屏障后唤醒逻辑是如何实现的

2.删栏循环使用是如何实现的

3.条件队列到同步队列的转换实现逻辑

https://www.processon.com/view/link/6197b0aef346fb271b36a2bf

AQS： 同步等待队列　　获取锁失败时　　入队，阻塞

独占锁的获取和释放
共享锁的获取和释放
公平和非公平
可重入
可中断

条件等待队列　　Condition

```
lock.lock();try{
    await() // 阻塞线程 释放锁　　获取锁
}finally{
   lock.unlock（）　　唤醒的是同步队列的节点绑定的线程
}
```

singalAll:　　前置步骤：条件等待队列转同步等待队列　　唤醒线程 ，重新获取锁

问题：
同步队列是后续的进双链表，条件队列是前面的进单链表么

Reentrantlock到底锁的是啥东西？是类吗？有点蒙了　　state
await和park都会阻塞么
条件等待队列为什么是单向列表

```
// 前半段：释放锁，进入条件队列，然后阻塞线程 （Thread0,Thread1）

//过渡阶段： 被其他调用singal/singalAll的线程(Thread2)唤醒 （前提：要在同步队列中）
// 调用singal/singalAll的线程(Thread2)：条件队列转同步队列，
// 可以在释放锁的时候唤醒head的后续节点所在的线程 (Thread0)

// 后半段：（Thread0) 被唤醒的线程获取锁（ 如果有竞争，cas获取锁失败，还会阻塞），
// ( Thread0) 释放锁（唤醒同步队列中head的后续节点所在的线程 （Thread1））
// 后半段的逻辑其实就是独占锁的逻辑
```